International Journal of Computer Science Engineering
and Information Technology Research (IJCSEITR)
ISSN (P): 2249-6831; ISSN (E): 2249-7943
Vol. 9, Issue 1, June 2019, 31-42

TRANS STELLAR
•Journal Publications •Research Consultancy

THE DRIVER FOR THE SCULL_OPEN DISCOVERY FUNCTION, READ / WRITE SCULL_READ / SCULL_WRITE FORA PROTECTED LINUX OS

OCHILOV NIZOMIDDIN NAJMIDDIN UGLI

State Testing Center under the Cabinet of Ministers of the Republic of Uzbekistan, Uzbekistan, Tashkent

ABSTRACT

In this article the principle of implementing a special-purpose device driver for secure Linux operating systems, using the example of a simple character driver is discussed. The main goal is to summarize and form the basic knowledge for writing future kernel modules. To interact with the equipment or perform operations with access to privileged information, the system needs a kernel driver. The Linux kernel module is a compiled binary code that is inserted directly into the Linux kernel, the internal and the least secure shell of executing instructions in the x86-64 processor. Here the code is executed completely without any checks, but at an incredible speed and with access to any system resources. Changing the kernel, you run the risk of losing data. The kernel code does not have standard protection, as in normal Linux applications.

KEYWORDS: Driver, Kernel, Opening, Reading, Writing, Closing, Kernel Level, Inode & Initialization

Received: Apr 06, 2019; **Accepted:** Apr 26, 2019; **Published:** May 07, 2019; **Paper Id.:** IJCSEITRJUN20195

Original Article

INTRODUCTION

The article discusses the principle of implementation of device drivers in secure operating systems (OS) Linux. The solution to this problem is relevant, since the creation of a secure OS causes problems with the interaction of devices. Linux provides a powerful and extensive API for applications, but sometimes it is not enough. A device driver is required to interact with equipment or perform operations. In order to ensure safe operation and safe handling of devices, a program is required [1]. The kernel communicates with devices through the appropriate drivers. A device driver is a collection of functions used to maintain it. One of the most important features of the Linux OS is the ability to dynamically load drivers. With this organization, the driver module becomes part of the kernel and can freely access its functions. In addition, a dynamically loaded driver may in turn be dynamically unloaded. If the driver is not explicitly unloaded, it remains permanently in the system until the next reboot [5].

If the module loads the system immediately after the start of the system starts, then this is the best failure scenario. The larger the code, the greater the risk of infinite loops and memory leaks. With carelessness, problems will gradually increase as the machine runs. In the end, important data structures and even buffers (intermediate data storage provided by software and intended to be transferred or copied between applications or parts of one application through cut, copy, paste operations) can be overwritten.

You can forget the traditional application development paradigms. In addition to loading and unloading a module, you can write code that will respond to system events, but it will not work in a certain sequence [1]. When working with the kernel, you can write an API, not the applications themselves.

MAIN PART

Sull_open. Much of the Linux system can be represented as a file. What operations are performed with files more often - opening, reading, writing and closing. Also with device drivers, you can open, close, read and write to the device (Figure 1).

Therefore, in the file operations structure, you can see such fields as: read, write, open, and. release are the basic operations that the driver can perform.

Driver File Management Subsystem

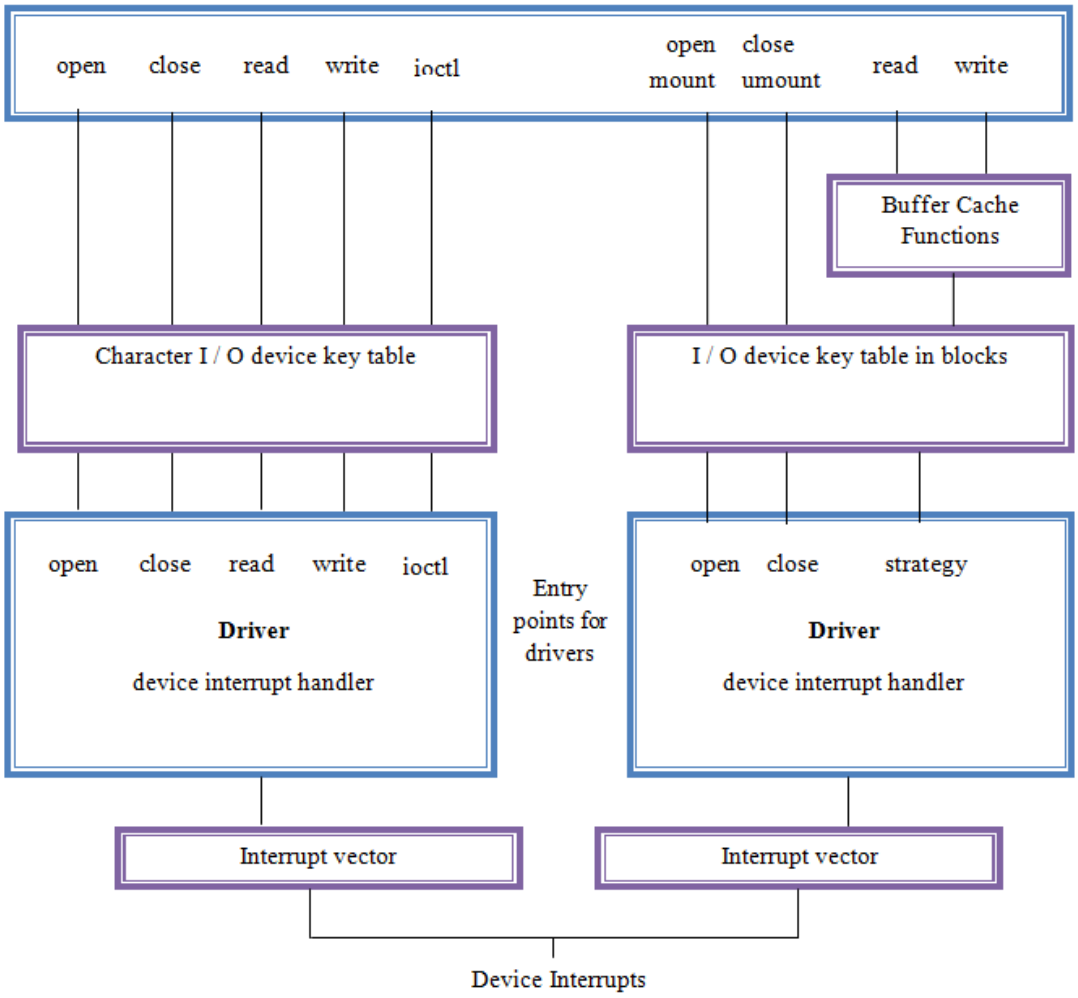

Figure 1: Driving Entry Point for Drivers

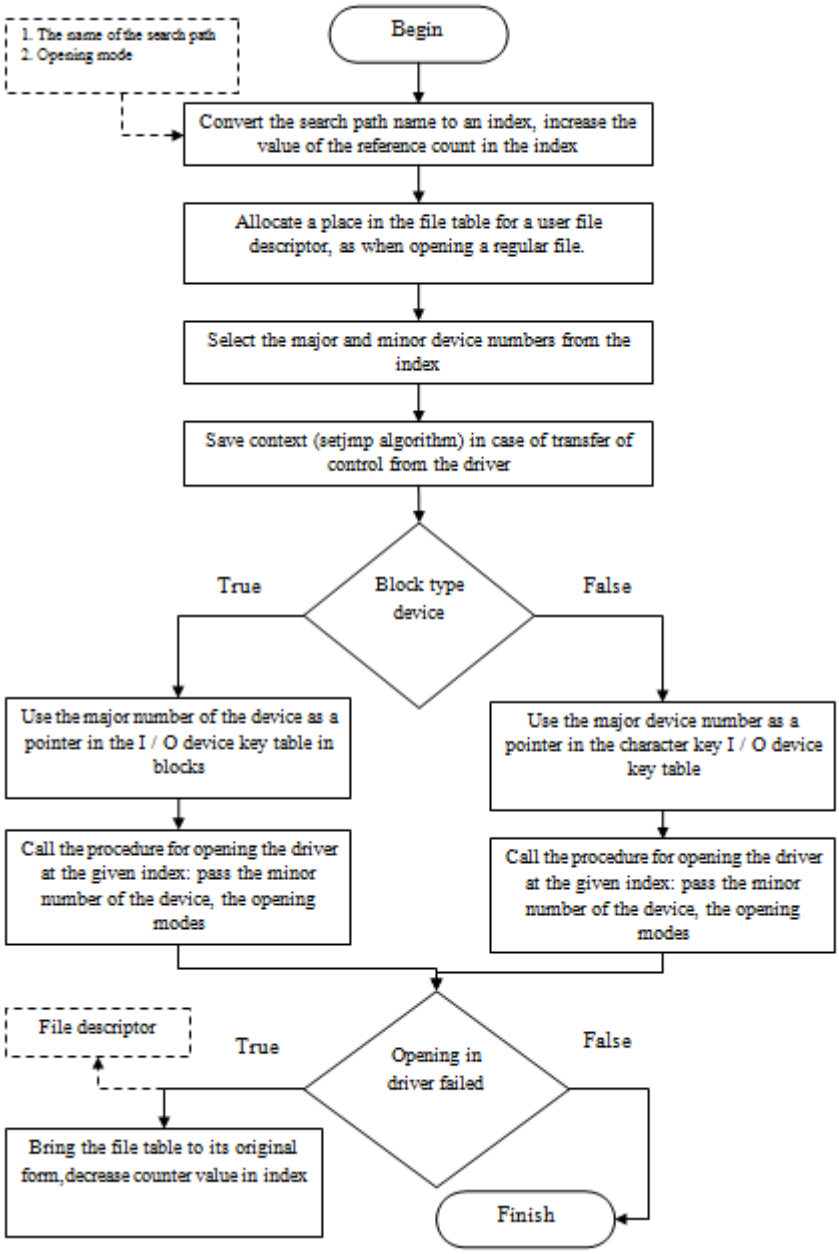

Figure 2: Block Diagram of the Device Opening Algorithm

```
int scull_open(struct inode *inode, struct file *flip)

{

        struct scull_dev *dev;

        dev = container_of(inode->i_cdev, struct scull_dev, cdev);

        flip->private_data = dev;

        if ((flip->f_flags & O_ACCMODE) == O_WRONLY) {

                if (down_interruptible(&dev->sem))

                        return -ERESTARTSYS;
```

```
                scull_trim(dev);
                up(&dev->sem);
        }
        printk(KERN_INFO "scull: device is opened\n");
        return 0;
}
```

The function Takes two Arguments

A pointer to an inode structure. An inode structure is an inode that stores information about files, directories, and file system objects.

A pointer to the file structure. The structure that is created by the kernel each time the file is opened contains the information needed by the upper levels of the kernel [1-3].

The main function of scull open is to initialize the device (if the device is opened for the first time) and fill in the necessary fields of the structures for its correct operation. Since the device does nothing, there is nothing to initialize.

Further, We will Execute Several Actions

```
dev = container_of(inode->i_cdev, struct scull_dev, cdev);
flip->private_data = dev;
```

In the above code, using container_of, we obtain a pointer to cdev of type struct scull_dev using inode-> i_cdev. The resulting pointer is recorded in the private_data field.

```
if ((flip->f_flags & O_ACCMODE) == O_WRONLY) {...
```

Further, if the file is open for writing, it is cleared before use and a message is displayed that the device is open (Figure 2).

scull_read. When a read function is called, several arguments are passed to it.

```
ssize_t scull_read(struct file *flip, char __user *buf, size_t count,
                loff_t *f_pos)
{
        struct scull_dev *dev = flip->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t rv = 0;
```

```
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        if (*f_pos >= dev->size) {
                printk(KERN_INFO "scull: *f_pos more than size %lu\n", dev->size);
                goto out;
        }
        if (*f_pos + count > dev->size) {
                printk(KERN_INFO "scull: correct count\n");
                count = dev->size - *f_pos;
        }
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum;
        q_pos = rest % quantum;
        dptr = scull_follow(dev, item);
        if (dptr == NULL || !dptr->data || !dptr->data[s_pos])
                goto out;
        if (count > quantum - q_pos)
                count = quantum - q_pos;
        if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
                rv = -EFAULT;
                goto out;
        }
        *f_pos += count;
        rv = count;
out:
        up(&dev->sem);
        return rv;
}
```

buf - is a pointer to a string, and _user reports that this pointer is in user space. The argument passes the user [2].

count – the number of bytes to read. The argument passes the user.

f_pos – bias. The argument passes the kernel. That is, when the user wants to read from the device, the read function (not scull_read) is called, while indicating the buffer where the information and the number of read bytes will be written.

```
if (*f_pos >= dev->size) {
        printk(KERN_INFO "scull: *f_pos more than size %lu\n", dev->size);
        goto out;
}
if (*f_pos + count > dev->size) {
        printk(KERN_INFO "scull: correct count\n");
        count = dev->size - *f_pos;
}
```

Checks

- If the offset is greater than the file size then read no longer works. An error is displayed and exits the function.
- If the sum of the current offset and the size of the data to be read is greater than the size of the quantum, then the size of the data to be read is corrected and report the message to the top.

```
if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        rv = -EFAULT;
        gotoout;
}
```

copy_to_user - copies data to buf (which is in user space) from the memory allocated by the kernel dptr-> data [s_pos] size count.

scull_write. The scull_write function is very similar to scull_read [4].

```
ssize_t scull_write(struct file *flip, const char __user *buf, size_t count,loff_t *f_pos)
{
        struct scull_dev *dev = flip->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
```

```
int item, s_pos, q_pos, rest;
ssize_t rv = -ENOMEM;
if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
item = (long)*f_pos / itemsize;
rest = (long)*f_pos % itemsize;


s_pos = rest / quantum;
q_pos = rest % quantum;
dptr = scull_follow(dev, item);
if (dptr == NULL)
        goto out;
if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
                goto out;
                memset(dptr->data, 0, qset * sizeof(char *));
}
if (!dptr->data[s_pos]) {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
                if (!dptr->data[s_pos])
                goto out;
}
if (count > quantum - q_pos)
        count = quantum - q_pos;
if (copy_from_user(dptr->data[s_pos] + q_pos, buf, count)) {
        rv = -EFAULT;
        goto out;
}
```

```
        *f_pos += count;
        rv = count;
        if (dev->size < *f_pos)
                dev->size = *f_pos;
out:
        up(&dev->sem);
        return rv;
}
```

Simplified Code

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/fs.h>
#include<linux/cdev.h>
#include<linux/semaphore.h>
#include <linux/uaccess.h>
int scull_minor = 0;
int scull_major = 0;
struct char_device {
char data[100];
} device;
struct cdev *p_cdev;
ssize_t scull_read(struct file *flip, char __user *buf, size_t count, loff_t *f_pos)
{
        int rv;
        printk(KERN_INFO "scull: read from device\n");
        rv = copy_to_user(buf, device.data, count);
        return rv;
}
ssize_t scull_write(struct file *flip, char __user *buf, size_t count, loff_t *f_pos)
```

```
{
        int rv;
        printk(KERN_INFO "scull: write to device\n");
        rv = copy_from_user(device.data, buf, count);
        return rv;
}
int scull_open(struct inode *inode, struct file *flip)
{
        printk(KERN_INFO "scull: device is opend\n");
        return 0;
}
int scull_release(struct inode *inode, struct file *flip)
{
        printk(KERN_INFO "scull: device is closed\n");
        return 0;
}
struct file_operations scull_fops = {
        .owner = THIS_MODULE,
        .read = scull_read,
        .write = scull_write,
        .open = scull_open,
        .release = scull_release,
};
void scull_cleanup_module(void)
{
        dev_t devno = MKDEV(scull_major, scull_minor);
        cdev_del(p_cdev);
        unregister_chrdev_region(devno, 1);
}
```

```
static int scull_init_module(void)
{
        int rv;
        dev_t dev;
        rv = alloc_chrdev_region(&dev, scull_minor, 1, "scull");
        if (rv) {
                printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
                return rv;
        }
        scull_major = MAJOR(dev);
        p_cdev = cdev_alloc();
        cdev_init(p_cdev, &scull_fops);
        p_cdev->owner = THIS_MODULE;
        p_cdev->ops = &scull_fops;
        rv = cdev_add(p_cdev, dev, 1);
        if (rv)
                printk(KERN_NOTICE "Error %d adding scull", rv);
        printk(KERN_INFO "scull: register device major = %d minor = %d\n", scull_major, scull_minor);
        return 0;
}
MODULE_AUTHOR("Name Surname");
MODULE_LICENSE("GPL");
module_init(scull_init_module);
module_exit(scull_cleanup_module);
```

CONCLUSIONS

The article presents the principle of implementation of a special-purpose of the device driver for secure Linux operating systems. The basics of an I / O device through mapped memory and macros used in memory allocation are discussed. As a practical example of allocating resources for an I / O device through the displayed memory, the code from the already debugged driver was given. The mechanisms for initializing and deleting devices in the Linux operating system kernel have been proposed and clarified. An optimized algorithm for initializing and deleting devices in the kernel of the Linux safe operating system has been developed, which makes it possible to optimize the running time of the algorithm by

reducing the number of unnecessary functions in the code. The algorithm is designed for protected Linux OS class 2A.

Thus, the article describes the procedure for working with kernel components. Using the acquired skills, you can develop your own kernel module and build security mechanisms in it.

REFERENCES

1. *Torvalds L., Diamond D. For Fun = Just for fun. - M.: EKSMO-Press, 2002. P. 288. ISBN 5-04-009285-7.*
2. *Love R. Linux kernel development = Linux Kernel Development. 2nd ed. - M.: "Williams", 2006. P. 448. ISBN 0-672-32720-1.*
3. *Rodriguez K. Z., Fisher G., Smolski S. Linux: ABC of the kernel. - SPb: "KUDITS-PRESS", 2007. P. 584. ISBN 978-5-91136-017-7.*
4. *Barret D. Linux: basic commands. Pocket guide. 2nd ed. - SPb: "KUDITS-PRESS", 2007. P. 288. ISBN 5-9579-0050-8.*
5. *Torvalds L. Linux Format = Linux format. - M.: EKSMO-Press, 2015. P. 228. ISBN 0-04-009183-1.*